CMR Development Guide

- 1
- 2 Getting Started
  - 2.1 Improving the Guide
  - 2.2 Prerequisites
    - 2.2.1 Required
    - 2.2.2 Recommended
  - 2.3 Running Code
    - 2.3.1 Clojure Components
    - 2.3.2 Ruby Projects
  - 2.4 Deployment
    - 2.4.1 Local
    - 2.4.2 Shared Environments
  - 2.5 Common Commands
    - 2.5.1 Clojure Project Commands
  - 2.6 Resources
  - 2.7 Project Overview
  - 2.8 Development Process
    - 2.8.1 Requirements
    - 2.8.2 Acceptance Criteria
    - 2.8.3 Sprints
    - 2.8.4 Scrums
    - 2.8.5 Earthdata Review Board
    - 2.8.6 Making Code Changes
  - 2.9 Technologies
  - 2.10 Tools
    - 2.10.1 Programming Languages
    - 2.10.2 IDEs
    - 2.10.3 Persistence Stores
    - 2.10.4 Continuous Integration
    - 2.10.5 Demos and Gorilla REPL
  - 2.11 Testing
    - 2.11.1 Unit Testing
    - 2.11.2 Integration Testing
    - 2.11.3 Performance Testing
- 3 Development Style Guide
  - 3.1 General Guidelines
    - 3.1.1 Small Tasks
    - 3.1.2 Understand, Design, Implement
    - 3.1.3 Pair Programming
    - 3.1.4 Code Reviews
    - 3.1.5 Conventions
  - 3.2 Testing
    - 3.2.1 Clojure Testing

Getting Started

Improving the Guide

This guide is a constant work in progress. If something is missing or confusing, please update the guide with better information.

Prerequisites

Required

The following items are required for developing the CMR.

1. *nix Style Bash Prompt
2. Java 1.7.0_25 or higher
3. Leiningen (http://leiningen.org) 2.5.1 or above.
    a. We recommend you install it using the download script instead of a package manager.
    b. Follow the above link. Click on the "lein script" link. Double click on a mac to get a menu and select "Save Page As..."
    c. Save the script using the name of "lein" to where you want to run it. (Using the format as txt is good.)
    d. Be sure to put the script in a place you want to run it from in the future and put the directory where the script lives in your path if it doesn't already exist. You may need to restart your shell or terminal. Test your path by typing "echo $PATH"
    e. Make the lein script executable.
    f. Run the script by typing "lein"
    g. To test the install type "lein repl".
4. RVM (http://rvm.io/)
5. Git
6. Vagrant (http://www.vagrantup.com/)

Recommended

The following libraries and tools are recommend. This guide assumes that you have installed the required and recommended libraries.

1. Mac OSX 10.8+
2. Homebrew (http://brew.sh/)
3. Sublime Text 2 or 3
    a. Configured following instructions here: https://github.com/jasongilman/SublimeClojureSetup

    > The ClojureHelpers.py file depends on indentation. Some developers have pasted this file without indentation inadvertently and spent a long time trying to figure out why commands like cmd+alt+r do not work. Take care to preserve the indentation when creating this file.

    b. Later sections will reference actions and keyboard shortcuts defined in the previous resource.

1. Mac OSX 10.8+
2. Atom development environment.
    a. Extra info
        i. The Atom manual is really good. http://flight-manual.atom.io/
        ii. I got all of the following information from 2 sites:
            1. https://gist.github.com/jasongilman/d1f70507bed021b48625
            2. https://git.earthdata.nasa.gov/projects/CMR/repos/editor-settings/browse/atom
    b. Download Atom
        i. Get the Atom download by clicking on this link: https://atom.io/
        ii. Then press the Download button (Download for Mac) to download the zip file.
    c. Install Atom
        i. Follow the install instructions in the atom flight manual: http://flight-manual.atom.io/getting-started/sections/installing-atom/
            1. For me my Mac asked for my password and I was able to install it easily.
            2. I was also able to type which atom and got the appropriate response.
    d. Configure Atom
        i. Install new packages
            1. Now that Atom is open in the Welcome Guide tab, click on Install a Package and then on Open Installer. The settings tab should appear.
            2. On the left side click on +Install if it isn't already selected.
            3. For every package to install do the following:
                a. In the search box type in the package name and click on the Packages button.
                b. Click the install button in the package box for that package name. (If a package is already installed the button won't be there).
                c. Here is the list of packages to install:
                    i. proto-repl - Clojure REPL, autocompletion, etc.
                    ii. proto-repl-charts - Graphs and Charts
                    iii. ink - Proto REPL dependency used for inline display and the REPL output.
                    iv. tool-bar - Proto REPL uses this to display a tool bar with options.
                    v. Parinfer - Handles parentheses and general Lisp editing.
                    vi. lisp-paredit - Used only for proper indentation on newline and indenting blocks of code. (Hopefully Parinfer will handle all of these in the future)
                    vii. highlight-selected - highlights selected keywords throughout an editor.
                    viii. set-syntax - Easily change syntax with the command palette.
                    ix. markdown-writer
                    x. atom-beautify - Formatting JSON, XML etc.

ii. Change Atoms' Package Settings - follow the instructions on this page: https://gist.github.com/jasongilman/d1f70507bed021b48625 starting from "Package Settings". Following is the process to follow if you need additional help.
    1. In the search box for packages type language-clojure and click the Packages button. This package comes pre-installed.
    2. Click in the language-clojure package box or click on the Settings button.
    3. In the Clojure Grammer section make the following changes:
        a. Auto Indent: unchecked
        b. Auto Indent On Paste: unchecked
        c. Non Word Characters: ()"':,;~@#$%^&{}[]`
        d. Scroll Past End: checked
        e. Tab Length: 1
    4. The file is saved when you make the changes.
    5. Then click on the +Install button for the next package. Follow the same process for the rest of the changes up until "Atom Settings"
    6. For the **lisp-paredit Indentation Forms: use the following:**

```
try, catch, finally, /^let/, are, /are\d/, /^def/,
fn, cond,
condp, /if.*/, /.*\/for/, for, for-all, /when.*/,
testing, doseq,
dotimes, ns, routes, context, GET, POST, PUT,
DELETE, context*, GET*,
POST*, PUT*, DELETE*, extend-protocol, loop, do,
case, with-bindings,
checking
```

iii. Atom Setting Changes
    1. For the Atom Settings, click on <> Editor on the left menu in the settings window and make the changes as follows: (These are main Atom Settings related to Clojure that are different than the default.)
        a. Auto Indent On Paste: unchecked
        b. Scroll Past End: checked
            i. Due to this autocomplete issue there is a lot of flashing from the autocomplete window that pops up. Scrolling down farther usually resolves the issue.
        c. Also set the Set "Preferred Line Length" to 100 (information from https://git.earthdata.nasa.gov/projects/CMR/repos/editor-settings/browse/atom)
    2. Add the following code block to the end of the styles.less file.
        a. On the far left menu of Atom, make sure the .atom folder has been expanded. If not expand it by clicking on it.
        b. The styles.less is the last file. Click on it to be able to edit it. Add the text of the code block to your file.

```
.proto-repl-repl::shadow .lisp-syntax-error
.region {
  background-color: rgba(0, 0, 0, 0)
!important;
}
```

        c. Once you have made the changes save the file in the mac file > save menu bar. Then close the window by clicking on the X in the window's tab.
    3. Add to the init.coffee file.
        a. Click on the file. Then copy the contents from this link into the end of your file: https://gist.githubusercontent.com/jasongilman/d1f70507bed021b48625/raw/08feba06ce68faccfd44e4b7ee683e09879bd2f8/init.coffee
        b. Also add the following after the previous addition.

```
################################################
################################################
#####
# Allow autocompleting to work correctly with
packages that take over the enter key
```

```
autocomplete_then_run = (command) ->
  editor = atom.workspace.getActiveTextEditor()
  autocompleted =
atom.commands.dispatch(atom.views.getView(edito
r), 'autocomplete-plus:confirm')
  if !autocompleted

atom.commands.dispatch(atom.views.getView(edito
r), command)

atom.commands.add 'atom-text-editor',
'jason:enter-clojure', ->
  autocomplete_then_run('lisp-paredit:newline')


################################################
################################################
#####
# Additional CMR specific Proto REPL commands

runTestsCmd = "(def all-tests-future
  (future (println \"Running all tests
future\")
          (cond
            (find-ns 'cmr.dev-system.tests)
            (do
              (taoensso.timbre/set-level!
:error)
              ((resolve
'cmr.dev-system.tests/run-all-tests)
{:fail-fast? true :speak? true}))

            (find-ns
'cmr.common.test.test-runner)
            ((resolve
'cmr.common.test.test-runner/run-all-tests)
{:fail-fast? true :speak? true})

            :else
            (clojure.test/run-all-tests))))"

atom.commands.add 'atom-text-editor',
'cmr:run-all-tests', ->
  protoRepl.refreshNamespaces =>
    console.log "Running the tests now."
    protoRepl.executeCode(runTestsCmd)

# Commands to set different log levels
atom.commands.add 'atom-text-editor',
'cmr:logging-level-debug', ->

protoRepl.executeCodeInNs("(user/set-logging-le
```

```
vel! :debug)")

atom.commands.add 'atom-text-editor',
'cmr:logging-level-info', ->

protoRepl.executeCodeInNs("(user/set-logging-le
vel! :info)")

atom.commands.add 'atom-text-editor',
'cmr:logging-level-warn', ->

protoRepl.executeCodeInNs("(user/set-logging-le
vel! :warn)")

atom.commands.add 'atom-text-editor',
'cmr:logging-level-error', ->
```

```
protoRepl.executeCodeInNs("(user/set-logging-le
vel! :error)")
```

c. Once you have made the changes save the file in the mac file > save menu bar. Then close the window by clicking on the X in the window's tab.

4. Add to the keymap.cson file.
    a. Click on the file. Then copy the contents from this link into the end of your file: https://gist.githubuserrcontent.com/jasongilman/d1f70507bed021b48625/raw/08feba06ce68faccfd44e4b7ee683e09879bd2f8/keymap.cson
    b. Also add the following after the previous addition.

```
# Proto REPL CMR specific
'.platform-darwin .workspace
.editor:not(.mini)':
  'ctrl-d': 'proto-repl:exit-repl'
  'ctrl-l': 'proto-repl:clear-repl'
  'alt-cmd-a': 'cmr:run-all-tests'
  'ctrl-, a': 'cmr:run-all-tests'

# Find this line in your keymap.cson file and
add an additional keymap
'.platform-darwin .workspace
atom-text-editor[data-grammar~="clojure"]':

  # Fixes a problem with lisp indent and
autocomplete combination
  'enter':          'jason:enter-clojure'
```

c. Once you have made the changes save the file in the mac file > save menu bar. Then close the window by clicking on the X in the window's tab.

e. Restart Atom.
f. Read and follow the following child wiki page we created about using Atom

Obtaining Code

1. Obtain a URS account. (https://urs.eosdis.nasa.gov)
2. Login into the ECC (https://earthdata.nasa.gov/ecc) and request access.
    a. Wait until access has been granted.
3. Login to the ECC and find the Common Metadata Repository in the list of projects and request access to that.
    a. Wait until access has been granted.
4. Run
    a. `git clone https://<username>@git.earthdata.nasa.gov/scm/cmr/cmr.git <local-directory>`

       replacing username and project-name to checkout a project.
       either delete <local-directory> if cmr is a desired directory to place the software, or replace <local-directory> with a path/directory name where the code will reside.
    b. cd into <local-directory>
    c. Setup profiles:
        i. cd into dev-system
        ii. rename profiles.example.clj to profiles.clj
        iii. Ask a CMR developer for the passwords and edit the profiles.clj file.
        iv. cd ..
    d. run dev-system/support/setup_local_dev.sh

Running Code

Clojure Components

There are multiple ways to run a Clojure based Component. They are listed below.

In the REPL

The REPL should be used to run components under active development. It provides an easy way to quickly refresh the current code on the file system and restart with the new code.

1. Run
   `(reset)`

   in the REPL or use cmd+alt+r keystroke.

Using Leiningen

Leiningen can be used to run a component.

1. Run
   `lein run`

   in the project directory along with any required arguments.

Using a Pre-built Jar File

A component can be run from an "uberjar", a jar file built that contains the component and all its dependencies.

1. Run
   `lein uberjar`

   to build the jar file.
2. Run
   `java jar target/<project-name>-<version>-standalone.jar`

   along with any required arguments.

Ruby Projects

TODO We should document the process for running Ruby projects once we have created some in the CMR.

Deployment

Local

Shared Environments

TODO Document our deployment process once it has been established.

Common Commands

Clojure Project Commands

The following table contains commands for Clojure projects using Leiningen.

Command	Description

lein repl	Starts a new REPL at the command line. This is mostly done within Sublime Text.
lein run	Runs a Clojure application. Invokes the main method in the project.
lein uberjar	Builds a Clojure application jar including all of it's dependencies
lein clean	Removes any built assets from a project.
lein ancient*	Lists any dependencies in a project that are out of date.
lein install	Builds and installs a Clojure library into the local Maven repository.
lein check-deps	Checks for out of date dependencies in a project; an alias for lein with-profile lint ancient
lein modules do clean, install, clean	Cleanup and re-install. Use this command when dependencies have changed.
lein with-profile docs generate-docs	Generate API documentation in markdown (.md) files. When making changes to documentation, run this.
Linting and Static Analysis Commands	
lein kibit	Runs the kibit tool against the project source code
lein eastwood	Runs the eastwood tool against the project source code using the following eastwood configuration: "{:namespaces [:source-paths]}"
lein lint	Runs a higher-order task combining compile, kibit, and eastwood
lein bikeshed	Runs the bikeshed tool against the project source code
lein yagni	Runs the yagni tool against the project souce code

In addition, the top-level CMR metaproject supports the following commands:

Command	Description
lein check-deps	Runs check-deps against all subprojects
Linting and Static Analysis Commands	
lein kibit	Runs the kibit tool against all subprojects
lein eastwood	Runs the eastwood tool against all subprojects
lein lint	Runs a higher-order task combining compile, kibit, and eastwood against all subprojects

---

*Note that not all projects have lein-ancient installed in their top-level dependencies. All do have lein-ancient installed in the lint profile, however; as such, the "lein check-deps" command is available to all projects.

Resources

- CMR Wiki - Common Metadata Repository Home
- JIRA - https://bugs.earthdata.nasa.gov/browse/CMR
- Git/Bitbucket - https://git.earthdata.nasa.gov/projects/CMR
- Clojure for Java programmers - https://www.youtube.com/watch?v=P76Vbsk_3J0 - The entire ClojureTV youtube channel is filled with a lot of great lectures and screencast
- Clojure For The Brave And True - http://www.braveclojure.com/ - You can either buy the hard copy of this or just read it for free online
- Clojure Docs - https://clojuredocs.org/ - The clojure.core functions are especially useful
- Clojure Style guide - https://github.com/bbatsov/clojure-style-guide

Project Overview

Development Process

Requirements

The input to the development process is requirements in the form of user stories. All user stories for a single phase of development are roughly estimated in points which are equivalent to half a perfect person day.

Acceptance Criteria

Tickets are entered into JIRA as tasks, user stories, or bugs. User Stories are customer facing functionality. Tasks are backend functionality. All User Stories must be entered with Acceptance Criteria that identify the definition of done for the ticket. The acceptance criteria outline what should be tested to mark the ticket as completed. A good format for acceptance criteria is "Verify that..."

Sprints

Work is done in sprints, which are 2 weeks long.

Beginning

Retrospective

A Sprint Retrospective of the previous sprint is held the first day of a new sprint. Each developer notes good things and places where improvements are needed. Up to 5 goals are selected for the new sprint. Notes are kept in the wiki on the retrospective. The retrospective just includes members of the development team.

Sprint Planning

The sprint is kicked off with a planning meeting on the first day of the new sprint, separate from the retrospective. The development team along with stakeholders attend the planning meeting. Demos of work completed during the previous sprint are shown first and then the work planned for the upcoming sprint is discussed.

The amount of work put in a sprint is equal to percentage velocity * the number of hours available. Velocity is based on past sprint performance and adjusted each sprint based on predicted number of interruptions and problems.

Example: 75% Velocity * 4 people * 10 days * 2 points/day = 60 points of work available for a sprint.

Work is selected from user stories and non-user story tasks for the sprint. User stories are estimated in detail at the beginning of the sprint in perfect person days which are equivalent to 2 points.

Middle

Work is done and hours are logged in JIRA. Every developer must enter work on issues. The developer assigned an issue is responsible for adjusting the number of hours left in an issue. If another developer contributes to an issue they enter their own work but do not adjust the number of hours left.

Work must be entered for the previous day prior to the start of the Scrum. The burndown chart in JIRA shows the progress of work completed vs worked as the sprint progresses.

End

Any issues not completed are held for the next sprint.

TODO add more information about versioning once we establish a process.

Scrums

Scrums are short meetings held everyday at 10:30am and last up to 15 mins. Each developer has a chance to note issues or blockers. Status should be limited to a small description of progress made and what is planned for that day. Discussions of issues should be held until after the scrum.

Earthdata Review Board

Once a week, generally on Thursdays directly following the leads meeting there is an Earthdata Review Board (ERB) meeting. See https://wiki.earthdata.nasa.gov/display/EDO/ERB+Process for details.

Making Code Changes

Code changes are made through the use of Pull Requests in Bitbucket. The process is described at a high level below.

1. A developer is assigned an issue in JIRA.
2. The developer creates a local feature branch in the related projects and pushes the branch to Bitbucket.
3. The developers makes their code change and pushes the change to the remote branch.
4. The developer submits a pull request and assigns 1 or more other developers to review their code.
5. The reviewers review the pull request in Bitbucket and make comments. They decline the pull request if there are required changes.
6. The developer fixes any issues and pushes the fixes.
7. The developer reopens the pull request in Bitbucket.
8. The reviewers re-review the pull request. If the code changes are satisfactory the code is merged into master.

This process is documented in detail on the wiki page Code Change Helper.

Throughout the above process the issue should be updated in JIRA. The issue should be moved to the In Progress column when work is started, In Review column when the pull request is assigned to reviewers, and then Done when the pull request has been merged. Note that the Done column has two sections: Submit for Test and Complete. If the issue can be tested, move to the Submit for Test section. If no testing is needed (i.e. a documentation update), move to the Complete section. Most tickets will need to be tested.

Updating ticket status in JIRA throughout the sprint is very important to give an accurate sprint status at any time.

Technologies

Concepts

Micro Services

The CMR is composed of small, decoupled services called Micro Services. Martin Fowler has an excellent explanation of Micro Services here http://martinfowler.com/articles/microservices.html

The CMR is using micro services for the following reasons and benefits:

- Decoupling - Separate services are naturally decoupled from each other. Coupling between services is more obvious in separate code bases is more obvious when it happens through remote procedure calls.
- Polyglot - Separate services can be implemented in different languages. Individual services can be rewritten in a new language without breaking other services.
- Simplicity - Each service in a Micro Service Architecture is smaller and easier to understand. This is a trade off for increased complexity in the large between services and more difficult deployments.
- Reduced Application Size - Multiple, small services vs a single large application is easier to maintain with a team of developers. It's easier to separate work and responsibilities.

REST

REST stands for Representational State Transfer. CMR APIs are defined in a RESTful manner where appropriate.

Immutability

Immutability means that data cannot change. Once a piece of data has been created it is fixed in that state forever. Mutability or change is represented as a series of immutable values. The CMR utilizes immutability in both the large (system level) and in the small (within a single application) to achieve different benefits.

Immutability at the System Level

The primary use of immutability at the system level is in the handling of new concept (collections, granules, etc.) metadata. Each new copy of a collection, granule, or other concept received from a Provider is treated as an immutable *revision*. Each metadata concept instance is uniquely identified by a CMR Concept Id (example: C12345-LPDAAC_ECS) and a revision id, a number incremented on each new revision.

The concept id by itself can refer to all revisions of that metadata or *the latest* revision of a concept. Once a metadata revision has been stored it will never change though it may be garbage collected to reduce space if it has been superseded by a newer revision.

The benefits of treating data immutably at the system level are the following:

Easier synchronization

Immutable data is easier to synchronize. Catalog item data is kept in multiple places including Elasticsearch, pre-generated metadata in different formats, and caches. Because revisions never change they can be synchronized once from the source Metadata DB and a *revision* will always be correct because it can never change in the source database. Revisions can be cached forever. Synchronization from the source is also idempotent. Revisions can be synchronized multiple times and they will always be overwrite with the same value.

Scalability

The synchronization idempotency provided by immutability also means there are no race conditions if the same revision is synchronized by concurrent processes. Whichever process wins by being the last will still synchronize the correct data for a single revision. This holds as long as synchronizing a single revision is atomic. Indexing and pre-generation of derived metadata can be arbitrarily scaled to many processes. No synchronization locks are required so the scaling is essentially infinite.

Flexibility

Identifying each metadata update as a revision allows us to introduce arbitrary stages between the time a revision has been received and when it becomes public. A received revision may need to go through QA or other asynchronous auditing processes before it is indexed for searching.

History

We can tell when and exactly what changes were made with each update to an item.

Immutability Within an Application

Immutability is also used within applications. This is a standard feature of Clojure but we should try to utilize immutability with other languages as well. Immutability within an application gives the following benefits.

Avoids Bugs

Calling other functions with immutable data helps prevent bugs. Functions called with mutable data could manipulate it or they could be refactored at some point to mutate it in an unexpected way. This can cause bugs if the caller isn't expecting the data to be modified.

Easier Parallelization / Safe Concurrency

Another source of bugs is concurrency errors from dealing with mutable data. Safe guards such as locks are not required when dealing with immutable data.

Easier to Write Pure Functions

Pure functions are functions that do not manipulate state. They take in existing data as values and return new data. Pure functions are easier understand (data in - data out) and to test since they are so simple.

Idempotency

Operations are idempotent if final state is the same whether it is executed 1 time or many times. The *result* of multiple idempotent operations must be the same but the *responses* can be different. For example the result of 1 or many delete operations is then same in that the item no longer exists. The response to 1 or many delete operations is different. The first delete operation is a successful response (HTTP 204). Subsequent delete operations return not found (HTTP 404). Operations in the CMR should be idempotent where possible.

Idempotency is beneficial in distributed systems where communicate can get interrupted or just *look like* it was interrupted. This requires the sender to determine if it should resend a message. Resending messages to idempotent operations is safe to do.

Polyglot

The CMR is designed to allow the use of multiple languages. The best language for any particular task can be chosen. The qualities of "best language" need to take into account the current skills of the team and avoid adding too many different languages to the mix. The use of a new language will be reviewed before it is accepted.

Tools

Programming Languages

This is a list of the programming languages currently in use on the CMR.

Clojure

Clojure (http://clojure.org) is a functional programming language for the JVM. Clojure was selected for use on the CMR due to its appropriateness for high performance and highly scalable tasks.

Resources

Clojure Cheat Sheet - A list of clojure commands, descriptions, and examples

Clojure Workflow

Patterns and Techniques for Large Scale Apps (Video)

Clojure Protocols

IDEs

The IDE used is up to the individual developer. Most CMR developers use Atom. More information on how CMR uses Atom here.

Build Tools

Leiningen

Leiningen (http://leiningen.org) is the build tool for Clojure projects.

Persistence Stores

Oracle

Oracle is the current database for the CMR. it is used for storing the metadata and other general database needs. The CMR should avoid becoming dependent on any particular database and especially Oracle. The customer may want to move off of Oracle eventually due to cost issues and Oracle's potential issues in cloud deployments.

Elasticsearch

Elasticsearch (http://www.elasticsearch.org) is a Lucene based search index. The CMR uses it for searching for metadata.

Rabbit MQ

Rabbit MQ (https://www.rabbitmq.com) is a message queue.

TODO a message queue is needed for the CMR. Rabbit MQ is the proposed message queue implementation because it is standards compliant, mature, and capable of providing distributed, durable message processing. We should update this section once we have an opportunity to evaluate it.

Continuous Integration

Continuous integration is done through Bamboo. Bamboo continuously builds master as well as the feature branches.

To check a CI build you must be on NASA VPN. Go to https://ci.earthdata.nasa.gov

CMR Simplified Build is master or any branches set up to build artifacts for deployment. CMR Feature Branch with Oracle are the builds for

individual branches. A build is triggered when code is pushed.

Demos and Gorilla REPL

Demos are done in Clojure code via Gorilla REPL. Demos have been split out into their own repo in the CMR collection, cmr-demos. In `cmr-demos/demos` you will see a demo file for each sprint. Part of task work is completing a demo.

To use Gorilla REPL:

1. Start a REPL in the `cmr-demos` cloned directory: `lein gorilla`
2. After Gorilla starts up, it will print a message to standard out letting you know the port that it's running on as well as the complete URL to the Gorilla CMR demo worksheet, e.g. http://127.0.0.1:12345/worksheet.html
3. Paste that URL into your browser
4. Open the menu in the top right corner and select Load a Worksheet
    a. Worksheets are usually labeled by sprint, so your demo would go in the worksheet for the current sprint
    b. There are many to choose from, but as you start typing (e.g., "demos/sprint") the choices will narrow
5. To create a demo, add a section for your task. Sections can be code or markdown (text). Add Clojure code and any description text needed.
6. To execute Clojure code in Gorilla REPL, press Shift-Enter.
7. Make sure to save often!
8. When finished, check-in the demo file and push the changes.

The menu has all of the commands needed and other demos serve as great examples. All of the commands have key bindings associated with them.

Testing

There are three kinds of tests in the CMR, unit testing, integration testing, and performance testing.

Unit Testing

Unit tests in the CMR are used to test individual "units" of code. A unit of code will be different depending on the language. A function is a unit of code in Clojure. An object is the unit of code in an Object Oriented language in Ruby. Unit tests test each unit in isolation.

There are two types of unit tests, example based tests, and property based tests. Example based tests start with some predefined example input, execute a unit with the example input, and then verify the results exactly match was was expected. Example tests are good at verifying the results of a predefined set of examples are exactly what was expected. Their limitation is that they can not verify the results of examples not tested.

Property based tests start with the definition of input for a unit. This definition describes all possible values that could be used. These definitions are described as "generators". The generators are used to define properties that must hold true for all input. When a property based test runs it randomly generates values using the generators and verifies that every property holds true for each value.

Property based tests test many more examples than unit tests. A combination of example based and property based unit tests should be used to provide good coverage.

Integration Testing

Integration Tests test multiple CMR services together. Integration tests can be tested at the system level (the whole CMR) or for an individual application. Individual applications can be integration tested if they do not depend on other CMR services.

Performance Testing

Performance is a critical part of the CMR. There are system level tests for performance.

TODO We should write more about the performance testing as it is developed.

Development Style Guide

CMR code is expected to be of high quality. That means it needs to be:

- **Correct** - Does it do what's it's supposed to do?
- **Understandable** - Can I read the code and understand what's going on?
- **Maintainable** - Is it easy to change?

Different practices contribute to those aspects of development. The following sections identify ways that we keep the code correct, understandable, and maintainable.

FIXME design, testing, code reviews, pair programming

FIXME clear separation of concerns, use of comments, design, code reviews pair programming

General Guidelines

The process of writing code contributes to writing better code.

Small Tasks

Tasks assigned and worked on should be small. Smaller problems are limited, easier to understand, and implement. It's harder to off track and waste time when solving a small problem. Tasks should be a maximum of 2 days and generally less than 1 day. If a task takes longer to complete break it up. There's no limitation on a task other than it must define an objective.

Understand, Design, Implement

The process for solving a problem is Understand, Design, and Implement. Understanding the problem includes research, thinking about the problem, and making and testing hypotheses. This could include writing prototype code to test hypotheses.

The next step is to design the solution. Design could include making diagrams, talking with another developer, and writing on the whiteboard. The output of the design stage *is a design*. A design will have some kind of artifact but we don't require any specific kind. Whatever works best to solve the problem is ok.

The final step is implementation. We emphasize that there are two steps before this because they are important. A good design backed by research produces a good implementation.

Pair Programming

We pair program when a problem warrants it. There are many benefits to pair programming but not everyone likes it and it's not appropriate for every type of problem.

You should pair program when ...

- … working on a task that defines a new area or component.
- … working on an unfamiliar part of the code base. Ideally you can pair with the person who wrote the code so you can understand how it's working faster.
- … working on a difficult task.
- … working on a task that is taking longer than normal.
- … you are stuck.

Code Reviews

All major and sometimes minor code changes are reviewed during CMR development. Reviewing code helps spot problems, ensures consistent style and patterns, and provides a social pressure to write better code. All members of the development team participate in reviewing code. The code reviewer should look for best practices, readability, bugs, and other values listed in the CMR Development Guide. Since everyone participates in code reviews it will spread knowledge of different parts of the system and encourage common code ownership.

Conventions

We should establish and use conventions. Conventions in software development are important to help understandability of a code base. Think hard before introducing a new convention or ignoring an existing one. Document a convention when establishing a new one.

Testing

This section describes testing best practices within the CMR. This builds on top of the information presented in the Project Overview Testing section above.

Clojure Testing

Unit Test Guidelines

Unit tests in Clojure is done with the Clojure libraries `clojure.test` (built into Clojure) and `clojure.test.check`, an additional property based testing library.

Test small areas.

Unit tests should test small areas of code, preferably single functions.

Test pure functions.

Pure functions are functions that modify no external state or rely on external state to execute. They operate only on the data passed in as arguments and return new data. These are the easiest kinds of functions to test.

Avoid mocking.

Mocking is hard to get right, makes test brittle, and often misses testing crucial things.

Refactor code to make it more testable.

If it's hard to test some area of code it could mean that the code is coupled with side effects like communicating with a database. Refactor the code to move logic into pure functions.

Integration Test Guidelines

Integration testing in Clojure is implemented with the built in `clojure.test`.

Start with an empty initialized system.

Every integration test should assume that no data exists in the system. Data that is needed for testing should be loaded as part of the test. This is most likely done with a fixture.

Integration tests cleanup after themselves.

Any modifications to the system should be undone by the integration test. This will prevent one integration test from breaking another test by leaving unexpected data around.

Test at the API level.

Integration tests should test an application or the whole system as a black box. They shouldn't manipulate underlying resources used by the application.

Choosing between Integration and Unit Tests

We should test as much as possible at the unit test level. Unit tests are easier to write, run faster, and will be narrowly focused so errors are found more easily.

- Unit Tests are best for:
    - … testing pure logic like data manipulation or data conversion.

        - … testing single functions.
- Integration Tests are best for:
    - … testing integration of components in an application.
    - … testing integration of applications in a system.

Coding Style Guidelines

Coding style guidelines are documented on the Wiki page Coding Style Guidelines.